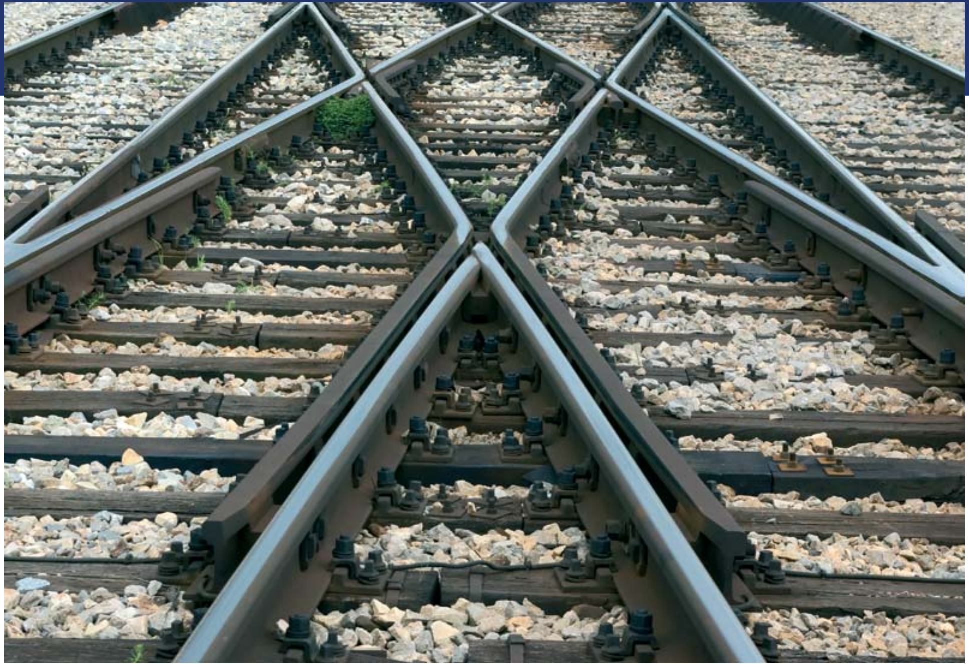
Parallel programming
Programming with OpenMP
Part 1



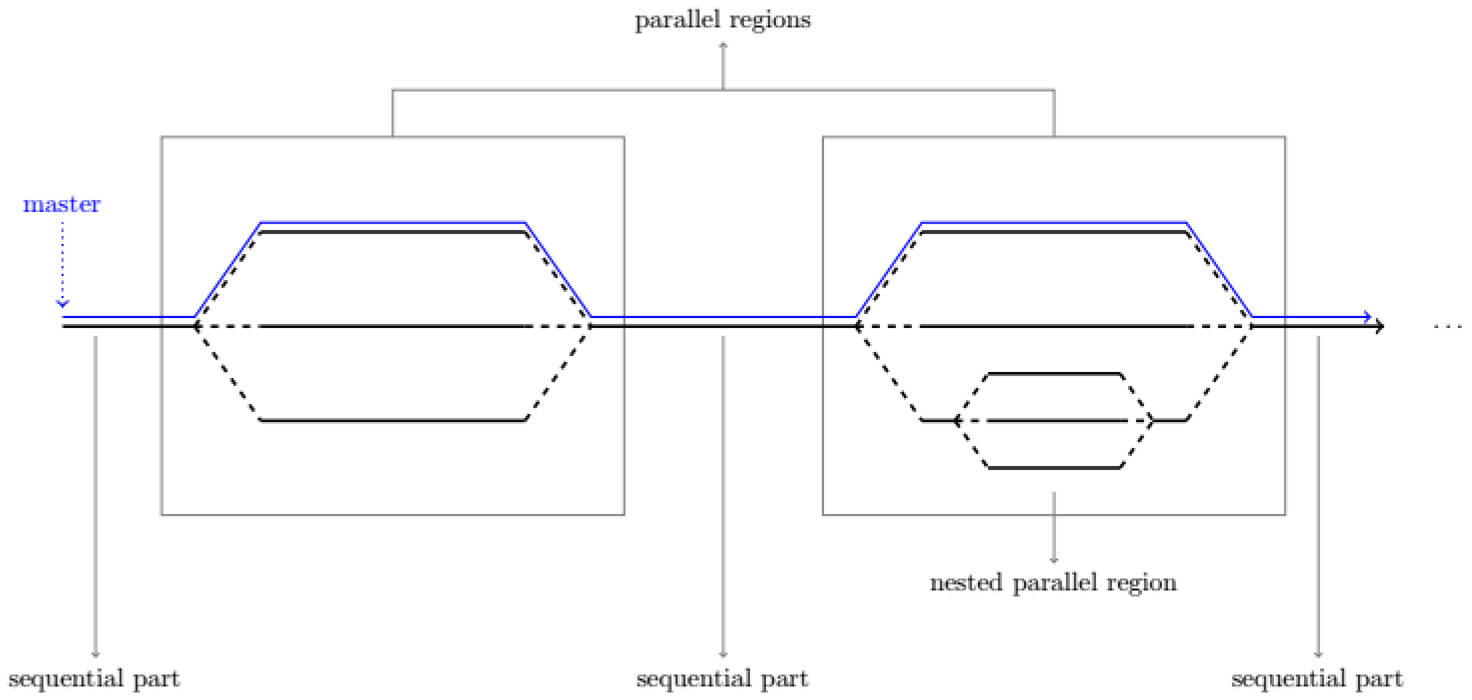
FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

Introduction to OpenMP

- OpenMP (Open Multi-Processing) provides constructs for parallel programming in C++, C, and Fortran on Linux, MacOS, and Windows.
- A sequential code is transformed to a parallel one by adding compiler pragmas, so if a compiler does not support OpenMP, the pragmas are skipped and the output is a sequential program.
  - **OpenMP manual: 1.3 Execution model:** *The OpenMP API is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library). However, it is possible and permitted to develop a program that executes correctly as a parallel program but not as a sequential program, or that produces different results when executed as a parallel program compared to when it is executed as a sequential program. Furthermore, using different numbers of threads may result in different numeric results because of changes in the association of numeric operations.*
- OpenMP is widely used in software like Blender, fftw, OpenBLAS, and eigen to accelerate computations.
- It is easy to use!

Execution model

- OpenMP program starts as a single thread only (*master thread).*
- It is executed sequentially until it reaches a **parallel region** defined by OpenMP pragma.
- At the entry of parallel region, new *team of threads* are created. Each thread executes concurrently with the others in order to share the work to be done.
- An OpenMP program alters between sequential regions and parallel regions.



Using OpenMP

- Include header file

```
#include <omp.h>
```

- Cmake (multi-platform)

```
find_package(OpenMP)
if (OPENMP_FOUND)
    set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
    set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} ${OpenMP_EXE_LINKER_FLAGS}")
endif()
```

- gcc

```
g++ -fopenmp ...
```

Hello world! In OpenMP...

`lab_codes/HelloWorld.cpp`

Runtime Library Routines

- **omp_get_num_procs()**

  Returns the number of processors that are available to the program
- **omp_get_num_threads()**

  Returns the number of threads that are currently in the team executing the parallel region from which it is called
- **omp_get_thread_num()**

  Returns the calling thread index within the current team
- ...

#pragma omp parallel

Creates team of threads and starts executing them in parallel

```
#pragma omp parallel
{
    cout << "This is thread " << omp_get_thread_num() << " speaking" << endl;
}

cout << "Parallel block finished" << endl;
```

Waits for threads to finish (barrier)

Output:

```
This is thread 0 speaking
This is thread 3 speaking
This is thread 2 speaking
This is thread 1 speaking
Parallel block finished
```

#pragma omp parallel

Creates team of 8 threads

```
#pragma omp parallel num_threads(8)
{
    cout << "This is thread " << omp_get_thread_num() << " speaking" << endl;
}
```

Output:

```
This is thread 0 speaking
This is thread 3 speaking
This is thread 6 speaking
This is thread 1 speaking
This is thread 2 speaking
This is thread 7 speaking
This is thread 4 speaking
This is thread 5 speaking
```

#pragma omp single

Block performed by single thread

```
#pragma omp parallel
{
    cout << "This is thread " << omp_get_thread_num() << " speaking" << endl;

    #pragma omp single
    {
        cout << "The single part was done by thread " << omp_get_thread_num() << endl;
    }
}
```

Output:

```
This is thread 3 speaking
The single part was done by thread 3
This is thread 1 speaking
This is thread 2 speaking
This is thread 0 speaking
```

#pragma omp sections

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            cout << "section 1, first: " << omp_get_thread_num() << endl;
            cout << "section 1, second: " << omp_get_thread_num() << endl;
        }

        #pragma omp section
        {
            cout << "section 2, first: " << omp_get_thread_num() << endl;
            cout << "section 2, second: " << omp_get_thread_num() << endl;
        }
    }
}
```

Each section is performed by one thread

Waits for threads to finish (barrier).
Can be changed by
`#pragma omp sections nowait`

Output:

```
section 2, first: 0
section 2, second: 0
section 1, first: 1
section 1, second: 1
```

#pragma omp critical

```
int sum;

#pragma omp parallel
{
    #pragma omp critical
    {
        cout << "Thread " << omp_get_thread_num() << " in critical region" << endl;
        sum += omp_get_thread_num();
    }
}

cout << sum << endl;
```

Critical region, performed by all threads but not at once (mutual exclusion)

Output:

```
Thread 1 in critical region
Thread 0 in critical region
Thread 3 in critical region
Thread 2 in critical region
6
```

#pragma omp barrier

Threads in team wait on the barrier

```
#pragma omp parallel
{
    cout << "Before barrier thread " << omp_get_thread_num() << endl;
    #pragma omp barrier
    cout << "After barrier thread " << omp_get_thread_num() << endl;
}
```

Output:

```
Before barrier thread 0
Before barrier thread 3
Before barrier thread 1
Before barrier thread 2
After barrier thread 1
After barrier thread 2
After barrier thread 0
After barrier thread 3
```

Example

- Write function for computing vector normalization. Split the vector into two halves, each is processed by one section.
  - You may use skeleton `lab_codes/VectorNormalization.cpp`

Sequential summing of matrix rows

```
vector<vector<double>> matrix;

vector<double> rowSums(matrix.size(), 0);
for (int i = 0; i < matrix.size(); i++) {
    for (int j = 0; j < matrix[i].size(); j++) {
        rowSums[i] += matrix[i][j];
    }
}

double sum = 0.0;
for (int i = 0; i < matrix.size(); i++) {
    sum += rowSums[i];
}
```

Parallel summing of matrix rows

Each iteration of for loop performed by a thread (in parallel) from the team

```
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < matrix.size(); i++) {
        for (int j = 0; j < matrix[i].size(); j++) {
            rowSums[i] += matrix[i][j];
        }
    }
}
```

A shorter code...

```
#pragma omp parallel for
for (int i = 0; i < matrix.size(); i++) {
    for (int j = 0; j < matrix[i].size(); j++) {
        rowSums[i] += matrix[i][j];
    }
}
```

Question: what happens if you write the pragma on the inner loop?

If clause

```
#pragma omp parallel for if(matrix.size() >= 10)
for (int i = 0; i < matrix.size(); i++) {
    for (int j = 0; j < matrix[i].size(); j++) {
        rowSums[i] += matrix[i][j];
    }
}
```

Threads are only created for large matrices. Small matrices are summed sequentially since it does not pays off to create threads.

Reductions

- Parallel aggregation of an expression, e.g., a sum

```
sum = rowSums[0] + rowSums[1] + rowSums[2] + ... + rowSums[matrix.size() - 1];
```

Operators:
+, *, -,
&, |, ^, &&, ||,
max, min

List of variables:
var_1, var_2, …, var_n

Useful for doing multiple reductions at once

```
double sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < matrix.size(); i++) {
    sum += rowSums[i];
}
```

Collapse

Collapse for loops into one for distribution of the work among threads

```
int numRows = matrix.size();
int numCols = matrix[0].size();

double sum = 0.0;
#pragma omp parallel for collapse(2) reduction(+:sum)
for (int i = 0; i < numRows; i++) {
    for (int j = 0; j < numCols; j++) {
        sum += matrix[i][j];
    }
}
```

Data sharing

```
int a = 10;
int b = 100;
#pragma omp parallel for
for (int i = 0; i < 10; i++) {
    int c = a * b + i;
}
```

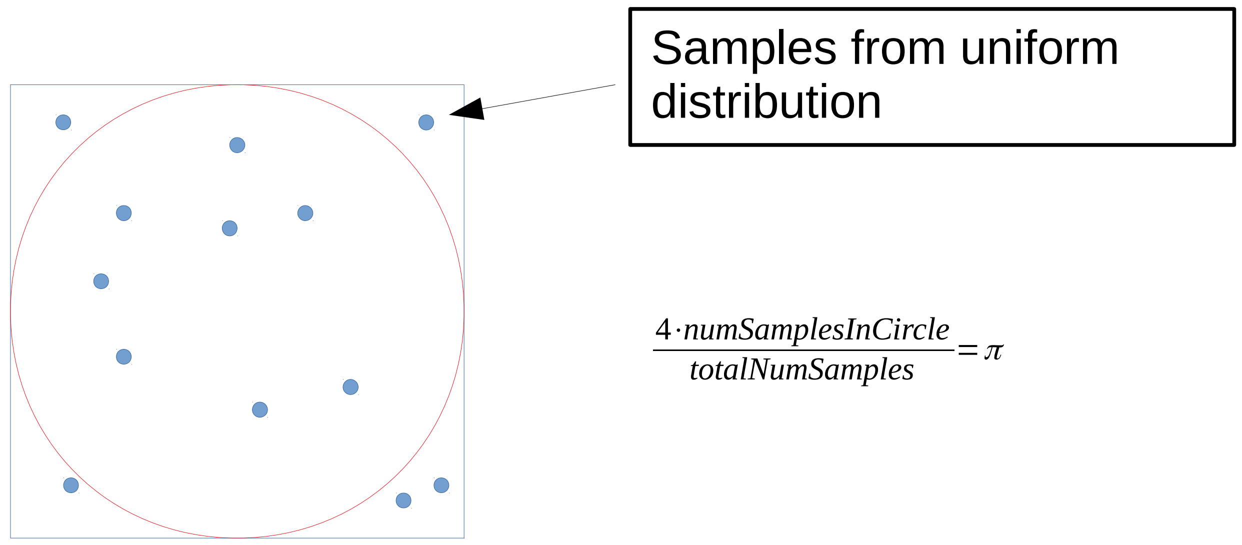
Shared among threads

Thread private

- The sharing can be stated explicitly as a clause
  - `private(a, b)`
    - Variables a and b are private to each thread (without global initialization)
  - `firstprivate(a, b)`
    - Variables a and b are private to each thread (with global initialization)
  - `shared(a, b)`
    - Variables a and b are shared among threads
- The default policy can be set to
  - `default(shared)`
    - By default, all the variables outside of the parallel section are shared
  - `default(none)`
    - The programmer must explicitly state the sharing policy of the variables

Examples

- Vector normalization using parallel for (reduction, critical section ...)
- Computation of pi using Monte Carlo



$$\frac{4 \cdot numSamplesInCircle}{totalNumSamples} = \pi$$